Applications Software Guidelines

Recommendations for Writing Power Friendly Software

Abstract

This paper provides recommendations for writing power friendly software focusing on Microsoft Windows* 95 application and driver developers. Also, information on a software tool, the Intel Power Monitor (IPM), developed by Intel that monitors system activity on computers running Windows* 95 and Windows* NT 4.0 is provided.

Introduction

With the introduction of Windows* 95, many software developers are switching to this relatively new operating system as the OS of choice for their mainstream development efforts. This fact, coupled with the increase of power managed systems, both portable and desktop varieties, makes it important that software developers provide power-friendly software to the consumer marketplace.

This paper describes important facts about Windows* 95 power management that application and device driver developers should keep in mind to make sure that their products allow Windows* 95 to perform efficient power management whenever possible. Information are organized in the following order:

1. Windows* 95 Environment
2. Single-Threaded Applications
3. Multi-Threaded Applications
4. Drivers
5. Optimizing the System
6. Summary

In addition, general information relating to Intel Power Monitor (IPM) is discussed.

Windows* 95 Environment

Windows* 95 is a multi-threaded system, with preemptive processing. This means that several programs can be running at the "same" time, each taking a slice of CPU time to perform its processing. In addition, each program can have multiple threads running within its process.

Each thread in the system is given a chance to run based on its priority value. High priority threads are given the chance to run first, with lower priority threads are given the chance to run when the other threads are waiting for system actions. Actions may be either a message in the thread's queue, or some object being signaled, such as a semaphore or a mutex.

In systems with Advanced Power Management (APM), Windows* 95 makes a CPU idle call to stop the CPU and conserve the power it would use. For this to occur under Windows* 95, all the threads in the system must be in a "wait" state. This wait state is entered when a thread is waiting for some action to occur. Actions may be either a message in the thread’s queue, or some object being signaled, such as a semaphore or a mutex. When one of these actions occurs, and the thread is given the opportunity to run, then the thread will continue to execute not allowing the power management activity to continue.

The thread tells the Windows* 95 OS what type of action it is waiting or by calling special APIs. For example, if the thread wants to wait for a message in its queue, it could call GetMessage. If the thread wants to wait for a semaphore to become signaled, it could call WaitForSingleObject. Additionally, if the thread wants ing the proper API with the proper arguments, the application tells the OS to halt execution of the current thread and cause it to wait for a specific occurrence before proceeding.

It is also possible for a thread to actively test whether a certain action has occurred, and then continue processing. An example of this is PeekMessage, which checks to see whether a message is in the thread's queue then returns immediately. Another example is WaitForSingleObject with the time-out period given as 0. These situations causes the thread not to enter a "wait" state and thus preventing Windows* 95 from performing any power management functions during the API call.

Single-Threaded Applications

Single-threaded applications can be power-unfriendly if written without accounting for the Windows* 95 behavior mentioned above. With simple single-threaded applications that use the following familiar GetMessage loop:

```
while (GetMessage(&msg, hwnd, 0, 0) == TRUE){
  TranslateMessage(&msg);
  DispatchMessage(&msg);
}
```

There is usually no problem, because the application enters the "wait" state each time that it calls GetMessage and only continues processing when GetMessage returns with a message.

However, consider the following loop:

```
done = FALSE;
do {
  if (PeekMessage(&msg, (HWND) NULL, 0, 0, PM_REMOVE)){
    if (msg.message == WM_QUIT){
      done = TRUE;
    }
    else {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
    }
  }
  else if (background_processing_required) {
    /* do some chunk of extra processing here */
  }
} while (!done);
```

Here it is very clear that the author intends to do some sort of processing while there were no messages available in the program's queue. This works because PeekMessage returns immediately, whether or not a message is actually waiting in the program's message queue. This code works fine, but it will keep the thread continuously active and prevent power management activity (idle calls) in Windows* 95.

It is possible that an application may require this type of intrusion on power management activity, but it is doubtful that an application will require this extra processing at all times while the program is active.

Fortunately, there are a couple of methods to maintain this type of functionality when necessary and, when it is not necessary, to let the power management activity continue normally.

Below is the same loop but with additional code, including a call to WaitMessage:

```
done = FALSE;
do {
  if (PeekMessage(&msg, (HWND) NULL, 0, 0, PM_REMOVE)) {
    if (msg.message == WM_QUIT) {
      done = TRUE;
    }
    else {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
    }
  }
  else if (background_processing_required) {
  /* do some chunk of extra processing here */
    if (done_with_extra_processing_until_next_message) {
      WaitMessage();
    }
  }
} while (!done);
```

While background processing is required, the application calls PeekMessage so it can handle both the extra background processing and Windows** messages. When the application determines that the background processing is no longer necessary, it calls WaitMessage, which enters a "wait" state until the program's message queue contains a message. Until a message is received, the application will not block any power management activity. Thus, the application is more power efficient because it still maintains its functionality when necessary, but allows power management to continue when it is through with its "background" processing.

There is another possibility that an application may be required to address. What if the application wants to perform this type of processing occasionally, say to check a status variable? In this case, calling WaitMessage would be inappropriate, because perhaps no message would be sent until the user moves the mouse or presses a key. This would not allow the program to periodically check the status variable. There

are a few solutions to this, the first is to set a timer at some interval so that the program receives a WM_TIMER message at various intervals. This causes WaitMessage to return, the PeekMessage would then retrieve the WM_TIMER message and the program would pass it on to the proper handler. Another way of performing background processing is demonstrated in the following code:

```
done = FALSE;
do {
  if (PeekMessage(&msg, (HWND) NULL, 0, 0, PM_REMOVE)) {
    if (msg.message == WM_QUIT) {
    done = TRUE;
    }
    else {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
    }
  }
  else if (background_processing_required) {
    /* do some chunk of extra processing here */
    if
(done_with_extra_processing_until_next_message_or_100ms) {
      MsgWaitForMultipleObjects( 0, NULL, FALSE, 100,
                                 QS_ALLINPUT );
    }
  }
} while (!done);
```

As in the previous example, this code return when the program receives any message in its message queue, due to the QS_ALLINPUT parameter. Unlike the previous example, however, this MsgWaitForMultipleObjects call also times out in 1/10th second (100 milliseconds) and continue processing, even if no message has been received. The MsgWaitForMultipleObjects API allows the caller to provide the timeout period in milliseconds. This lets the program "gain control" periodically to do some kind of processing. It also allows the system to perform power management while waiting for the timeout to occur. However, it is important to note that this does not provide processing at regular timed intervals. This is because the MsgWaitForMultipleObjects call will also return once a message is received, which may be quite frequent at times. It also depends on the priority of the application relative to other applications currently running.

Probably the best method for performing background processing is to spawn a separate thread that performs the desired action. In this case, when the thread finishes its work and terminates, the main thread doesn't have to worry about calling WaitMessage or MsgWaitForMultipleObjects, since the spawned thread handles the background processing itself. This simplifies the main thread's code since the background processing is not under its direct control. For more information on multi-threaded applications see the next section.

Again, it is important to design an application to use the processing power that it needs, but to allow the operating system to manage power when the application does not need full processing power. Test code and real-life examples have shown that savings of several watts can be obtained by using these techniques when the system is not performing any CPU-intensive processing.

Multi-Threaded Applications

Multi-threaded applications can get quite complex and, as such, need lots of care in their design, implementation and testing. Fortunately, with regard to power management efficiency, they are not much more complex than single-threaded applications.

First, it is important to notethat the previous discussion about single-threaded applications and power management is also relevant to multi-threaded applications. Everything that was mentioned in the single-threaded application section applies here. The only real difference is that there are more considerations and possibilities with multi-threaded code. Each thread in the process, must somehow interact with the other threads. Normally, this can be done by posting messages to a thread, or by using objects such as semaphores, events, mutexes and critical sections. Usually these objects are used for synchronizing code, allowing one thread to control when another thread executes.

Like a single-threaded application, a multi-threaded application needs to consider when the entire process (all threads in a multi-threaded application) is in a "wait" state in order for Windows* 95 power management activity to occur. For example, if all threads are in a WaitMessage waiting for a message to be posted to its message queue, then power management activity will occur. However, if just one of the threads is executing a PeekMessage loop, no power management activity will be performed.

The APIs available to Windows* 95 applications for waiting for objects can have the same characteristics as the GetMessage API, if passed with the correct

arguments. Each of these "wait for object" APIs has a parameter designating a timeout in milliseconds. This timeout parameter can be zero, in which case the object is tested and the call returns immediately. If the timeout value is INFINITE, only the object becoming signaled can force the call to return. Other values indicate the call will timeout if the object isn't signaled before a specified time elapses. The list of "wait for object" APIs are:

- MsgWaitForMultipleObjects
- WaitForMultipleObjects
- WaitForMultipleObjectsEx
- WaitForSingleObject
- WaitForSingleObjectEx

Like GetMessage, all these APIs will allow Windows* 95 to idle normally while the API is waiting for an object to become signaled, that is, when the timeout value passed is not zero. If the timeout is zero then the function returns immediately without allowing power management activity. Therefore, use a timeout value of zero only when necessary. Wherever possible try to use an INFINITE timeout value, otherwise, even small values allow the system to idle

Drivers

Like applications, drivers in Windows* 95 must also be written carefully to make sure they don't prevent power management from occurring in the system. There are two types of drivers in Windows* 95: ring 3-drivers (.DRVs) and ring-0 drivers (.VxDs). DRVs are a kind of glorified DLLs, so the same rules apply to them as to applications.

VxDs, on the other hand, have a completely different set of APIs available to them. At least a few of these APIs have been proven to prevent the power management mechanism in Windows* 95. Among these are the BlockOnIdle and Wait_Semaphore APIs.

Many Windows*-based VxDs use the VMM call pairs _BlockOnID / _SignalID and Wait_Semaphore / Signal_ Semaphore for thread synchronization. Generally, the signaling of a thread occurs soon after it blocks. However in some cases, the time between a thread blocking and its signaling can be very long, sometimes lasting minutes or hours.

While the Wait_Semaphore call does include a Block_Thread_Idle flag that tells the Windows* system to "consider the thread idle when it blocks on the semaphore," the _BlockOnID call does not have such a flag. Therefore, threads that use the _BlockOnID call are considered non-idle when it blocks on the ID.

A VxD should not call _BlockOnID or Wait_Semphore without the Block_Thread_Idle flag set unless there is a compelling reason for not allowing the Windows* system to go idle. Using these calls unnecessarily causes system performance problems. The Windows* scheduler checks to see if there are any outstanding non-idle blocking threads before it allows background driver processing to occur, consuming CPU time and taking time away from other processes.

Many power management methods used on laptop and notebook computers are based on the system going idle when there is no processing to do. A VxD using the _BlockOnID or Wait_Semaphore (with the Block_Thread_Idle flag cleared) calls unwisely will make the system appear busy to power management software, resulting in excessive power consumption and shortening the time that the user can run the system on battery power.

In the future, the Windows* system will make more and more use of idle time to do background processing, which is designed to optimize system performance. VxDs that do not allow the system to go idle will adversely affect the performance of these techniques.

All these problems can be avoided by calling _BlockOnID only when the thread will not need to block for an extended period of time or by using the Wait_Semaphore with the Block_Thread_Idle flag set.

VxDs also have the option of registering for idle notification by calling the Call_When_Idle API, to which they pass a callback address. This allows the VxD to perform processing whenever the system goes into an idle state. When the VxD's idle callback handler is called, the normally returns from the handler with the carry flag set. This tells Windows* 95 that it is okay to enter the idle state. However, if the VxD returns from the idle callback handler with the carry flag clear, this forces Windows* 95 to stop the idle notification callbacks and to skip any power management activity. Returning with the carry flag clear from an VxD idle callback handler must be done with great care to ensure that power management is performed whenever possible. It would not be acceptable to write a VxD such that it always prevents power management from occurring in the system.

Optimizing the System

Besides software, a known setting in the system can effect how efficient power management works in certain situations. The KeyIdleDelay setting in the [386Enh]

section of SYSTEM.INI controls the non-idle time after a keystroke while in a DOS box. By default, this value is set so that after pressing a key, the system will remain non-idle for about 1/2 of a second. This causes the system to remain non-idle while typing DOS commands or editing a document within a DOS box.

Setting the KeyIdleDelay equal to 0.005 can help the system to idle normally while typing in a DOS box and save power. However, don't use zero for this setting, because this may cause a symptom where the ALT key is not recognized at all times.

Summary

Writing power-friendly applications for Windows* 95 is not difficult. It is important to design any application carefully, considering the trade off between performance and power conservation. The things to keep in mind while designing a Windows* 95 application are:

- All threads in the system must be waiting for "messages" and/or "object(s) to be signaled" for Windows* 95 perform idle power management. One multitasking-unfriendly program running can ruin any efforts to conserve power in other applications.
- Applications should make use of the following APIs wherever possible. These routines essentially wait within the Windows* 95 kernel and allow idle power management to occur:
    - GetMessage
    - MsgWaitForMultipleObjects
    - WaitForSingleObject
    - WaitForMultipleObjects
    - WaitForSingleObjectEx
    - WaitForMultipleObjectsEx
    - WaitMessage
- When background processing is required, it is better to spawn a separate thread than have the main thread worry about performing two different tasks. This makes writing power-friendly code much easier.
- It is highly recommended that the WaitForObject APIs be used to perform synchronization between threads in a process. Wherever possible, use INFINITE for the timeout value to these APIs. If you can't use INFINITE, then try not to use zero; use as large a value as possible.
- When an application needs processing power it is okay to use it. But if or when an application no longer needs to fully utilize the processor (such as when sitting idle at a dialog box), be sure to use the above APIs to ensure power consumption is reduced.
- When writing VxDs, use the BlockOnID/SignalID APIs sparingly, because the _BlockOnID call prevents power management from occurring until the paired SignalID call.
- When writing VxDs that use the Wait_Semaphore/Signal Semaphore APIs, try to use these calls with the Block_Thread_Idle flag set. Otherwise, the blocked thread is considered non-idle and prevents power management from occurring until the semaphore is signaled with a Signal_Semaphore call.
- When writing VxDs that register with Call_When_Idle, make sure that the idle callback routine sets the carry before returning whenever possible. Otherwise, power management in the system will not be utilized.

Intel Power Monitor (IPM)

IPM is a power management tool for your Pentium® processor computer was developed by the Mobile Technology Lab at Intel.

What Is It and What Does It Do?

The Intel Power Monitor is a software tool that monitors system activity on computers running Windows* 95 and Windows* NT* 4.0 to provide information about software that may be wasting power.

Users of Pentium processor notebook computers can benefit from the Intel Power Monitor's power saving features. Intel Power Monitor can extend the battery run time of your notebook computer.

The Intel Power Monitor has been designed to facilitate the Independent Software Vendor's creation of power friendly software.

Optimizing Your Pentium® Processor Notebook Computer For Battery Run Time

In this section, we will discuss information on optimizing Pentium processor notebook computers for battery run time.

What is Power Management and Why Do I Have It?
The power management software built into your Pentium® processor notebook system and in Windows* 95 conserves power to extend battery life. This is done by shutting down various portions of the system when the system determines that user activity does not justify keeping them active. User activity is usually detected by the system monitoring keystrokes, mouse clicks, and mouse movement. Actions that power management may take include: spinning down the disk drive, dimming the display or turning it off, halting the CPU when the system is idle, and several levels of shutting down the entire system.

The power management software makes decisions based on user activity and power management policy values. These values are specified by the user, the notebook computer manufacturer, and the operating system vendor.

You can modify some of these parameters to allow the power management software to act more quickly or to delay action in shutting down portions of the notebook computer.

Longer Battery Run Time Is Obtained By Reducing Responsiveness
Turning off your notebook computer will maximize your battery life, but it makes getting work done a lot harder.

To maximize battery run time, choose aggressive power management parameters. For example, have power management shut down the disk and the display after one minute of user inactivity. If you normally view the screen for extended periods of time between using the keyboard or mouse, the display may turn off as you are reading it. If you find this distracting, increase the display's shutdown timeout value to more closely match your work habits.

As you use software applications on your notebook computer, be aware of interactions between the power management parameters you have selected and your software's activity. For example, if you have selected a three minute shutdown for the disk drive and your word processor has its autosave feature configured to save your work to the disk every three minutes, the disk drive may shut down for only brief periods. This could cause the disk to consume more power on average than if it were never shut down at all.

On the other hand, waiting the five seconds or so for the disk drive to spin up may disrupt the way you prefer to work. Having the display shut down before you have finished reading the page can be annoying. Don't be afraid to experiment with these settings until your notebook computer works as you want it to work.

Increasing Battery Run Time Of Your Notebook Computer:

- Turn off the autosave feature in your applications (or reduce the frequency of saves) to allow the disk drive to remain powered down for extended times. On a notebook computer, the RAM is powered from the battery rather than the A/C mains. Therefore, power glitches should not be a problem. Each time autosave saves the file, the hard drive must be powered up using power at a high rate to spin up, then it idles at lower rate until power management spins it down again several minutes later. The disk may run for five or more minutes although it performed useful work for only a few seconds of that time.
- Use your system RAM to keep commonly used software open. Switching among applications being held in memory, rather than loading them from disk, will keep your disk drive powered down for longer periods of time.

- As you become more familiar with your software, turn off wizards in your applications. If your application is reformatting text, checking spelling, checking grammar , recalculating spreadsheets, etc. In the background, the application is unnecessarily keeping the CPU busy. Additionally, wizards may access the hard disk to look at data or load additional software required to perform their function. Invoke those features when you need them, not when the application decides to do so.
- Use your display wisely to conserve your battery. Keep your backlight at the minimum brightness that you find comfortable to use.
- Rather than that nice wallpaper, use a solid color for the desktop background. Each time you move, resize, open or close windows, the background wallpaper must be redrawn. More CPU effort (and thus, more battery energy) is required to redraw complex images than to redraw a solid color.
- Use a blank screen as your screensaver. Screensavers that move calculations that some screensavers perform calculations to provide sound effects. While screensavers are nice perform no useful work, so turn them off or choose a blank-screen screensaver to hide the display and provide password security.
- Set your notebook computer's power management features to the most aggressive that you can tolerate.
- The Windows* 95 Plus Pack* installs the System Agent in your the System Agent, such as monitoring your system for backups and automatic disk defragmentation, require software to be running all the time. This unnecessarily consumes power and accesses the disk. Perform these functions manually when your system is attached to power lines.
- If you are leaving your computer for a while to do something else, suspend your notebook computer to save power.
- Choose a system with Smart Battery.

Windows* 95 Power Management Configuration

Control Panel — Verifying Power Management Support

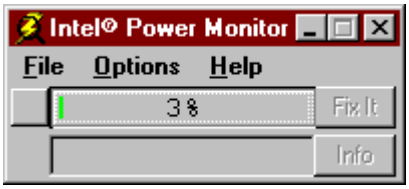
Perform the following steps to ensure that Windows* 95 is properly configured for Power management on your notebook:

- Click the Start button on the taskbar.
- Slide the selector up to Settings, choose Control Panel.
    - Double-click the System icon, and then click the Device Manager tab.
        - Click the + sign left of the System Devices icon to expand its display.
        - If Advanced Power Management support is visible, double click it to display properties.

          Device Status should indicate that the device is working properly. If Management support is not visible, it may have not been installed on your system. See Microsoft notes Q135136 (*www.microsoft.com/kb/articles/Q135/1/36.htm*), Q153395 (*www.microsoft.com/kb/articles/Q153/3/95.htm*), and Q137402 (*www.microsoft.com/kb/articles/Q137/4/02.htm***)**.
        - Click the Settings tab and make sure Enable power management support is checked.
    - There should be a battery icon displayed with the title Power.
        - Double-click it to display current settings.
        - Choose Advanced Power Management to enable Windows* 95 to perform power management on your notebook.

Intel Power Monitor — Features and Operation

The Intel Power Monitor has a simple set of menus and controls. You can activate the Intel Power Monitor program by starting it from the Windows* Run menu or by double clicking it in Explorer. Along with the standard title bar and menu items, the Intel Power Monitor displays a bar gauge (power bar) that graphically represents the amount of non-idle system activity in your system. This display updates approximately once per second.



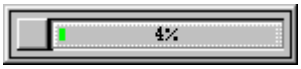
The color bar indicates CPU activity due to the software currently active.

- Green indicates low activity.
- Yellow indicates that the system is active 75% or more of the time.
- Red indicates that the processor was unable to go idle during the last measurement period.

To the right of the power bar is a Fix It button that indicates that the Intel Power Monitor has detected a behavior that it can correct (when not grayed out). Underneath the power bar is an area that displays the name of the offending application. You can click the Info button to obtain further information about the reported application.

Superimposed on the power bar is the percentage of non-idle CPU activity during the last measuring period.

Mini Bar (Simulation):



Intel Power Monitor also has a small button to the left of the power bar that you can click to change modes from Normal to a Mini Bar to reduce the screen area used by the display. The Mini Bar initially locates itself on the upper portion of your screen. You can position the Mini Bar anywhere on your screen by placing the cursor over the power bar area, pressing and holding the left mouse button, and then dragging the Mini Bar to the location of your choice.

System Activity and Power Management

Power management software extends battery life by shutting down the disk, display, processor, and other parts of your notebook computer when those parts are not performing useful work. Power management software usually resides in the system BIOS, but may reside in the operating system.

High non-idle CPU activity is not necessarily bad. For example, when a spreadsheet recalculation occurs, high processor activity is expected to complete the recalculation in a short time. However, if your software application is waiting for you to provide some input (e.g., choosing a font or filename in a dialog box), the processor should be inactive. The Intel Power Monitor will indicate this low activity by displaying a short green bar and a percentage value, typically under 25%. If you see a red bar instead, your application may be wasting power.

By fixing software that wastes power, the software vendor allows the power management features of your notebook computer to extend battery run time by reducing power consumption. The primary characteristic of a power-wasting application is a Power Monitor display that remains in the red for extended periods of time, when the application is waiting for you to type a character or choose an option with the mouse.

Fix It Button

If an application does not allow your system any idle time and the Intel Power Monitor has determined that the cause is fixable, the Fix It button will be enabled. Clicking this button causes the system to rest for brief periods of time. This rest time will be so short that you will not notice any difference in the way your application behaves. Behind the scenes, however, your battery life will be extended.

If you fix an application, you can undo the fix later (See the Fix Table section in Power Monitor's online help). The fixed application will display "Intel Power Monitor: FIXED" in its title bar, and the Intel Power Monitor itself will display the number of fixed applications in its title bar.

IPM Professional Features for ISVs

The Intel Power Monitor includes a "professional" mode that enables advanced features for use by ISVs in analyzing and fixing power-unfriendly software. The "professional" mode provides the following advanced features:

- Configurable Threshold Parameters
- Moving Performance-Trend Charts
- Enhanced Event Logging
- Application Debugging Support
- Extended Information Display

Independent Software Vendors can download Intel Power Monitor for Windows* 95 and Windows* NT from Intel web site:

www.intel.com/ial/ipm/download.htm.

You can contact us for instructions on configuring the Intel Power Monitor for "professional" mode operation by e-mailing *owner-ipm@mailbag.intel.com*. Please include your name, company name, address, and phone number. Further, we could include you in our list of

power friendly applications database if you inform us when your software is power-friendly.

System Requirements

The Intel Power Monitor runs on Pentium® processor and Pentium® Pro processor-based computers running Windows* 95 (with a minimum of 8 megabytes system RAM) and Windows* NT (with a minimum of 16 megabytes system RAM).

References

For further information, please refer to Intel developer web site *http://developer.intel.com/* or email *owner-ipm@mailbag.intel.com*.